

# Minimum-Link Paths Revisited

Joseph S. B. Mitchell\*      Valentin Polishchuk†      Mikko Sysikaski†

## Abstract

A path or a polygonal domain is *C-oriented* if the orientations of its edges belong to a set of  $C$  given orientations; this is a generalization of the notable rectilinear case ( $C = 2$ ). We study exact and approximation algorithms for minimum-link  $C$ -oriented paths and paths with unrestricted orientations, both in  $C$ -oriented and in general domains.

Our two main algorithms are as follows:

A subquadratic-time algorithm with a non-trivial approximation guarantee for general (unrestricted-orientation) minimum-link paths in general domains.

An algorithm to find a minimum-link  $C$ -oriented path in a  $C$ -oriented domain. Our algorithm is simpler and more time-space efficient than the prior algorithm.

We also obtain several related results:

- 3SUM-hardness of determining the link distance with unrestricted orientations (even in a rectilinear domain).
- An optimal algorithm for finding a minimum-link rectilinear path in a rectilinear domain. The algorithm and its analysis are simpler than the existing ones.
- An extension of our methods to find a  $C$ -oriented minimum-link path in a general (not necessarily  $C$ -oriented) domain.
- A more efficient algorithm to compute a 2-approximate  $C$ -oriented minimum-link path.
- A notion of “robust” paths. We show how minimum-link  $C$ -oriented paths approximate the robust paths with unrestricted orientations to within an additive error of 1.

## 1 Introduction

Minimum-link problems arise in motion planning with turn costs, in line simplification, guarding applications, VLSI, wireless communication, and other areas. An instance of the problem is specified by an  $n$ -vertex polygonal domain  $P$  with  $h$  holes, and two points  $s, t \in P$ ; the goal is to find an  $s$ - $t$  path with the fewest edges (links). In the query version of the problem, the goal is to build a data structure (link distance map) to efficiently answer link distance queries with  $s$  fixed.

The algorithm of Mitchell, Rote and Woeginger [25] computes a minimum-link path in  $O(n^2\alpha^2(n)\log n)$  time, where  $\alpha$  is the inverse Ackermann function. It was believed that a faster algorithm is possible (e.g., in [5, p. 263] the result of [25] is called “suboptimal”). Nevertheless, the only previously known lower bound, also due to [25], was  $\Omega(n\log n)$ . The same bounds for the *rectilinear* case are given in [5, 22]. Also, no approximation algorithm was previously known.

In this paper (Section 2) we give a subquadratic-time  $O(\sqrt{h})$ -approximation algorithm for the minimum-link path problem. We also observe (Theorem 2.1) that finding the exact solution is

---

\*Department of Applied Mathematics and Statistics, Stony Brook University.  
joseph.mitchell@stonybrook.edu

†Helsinki Institute for Information Technology, Department of Computer Science, University of Helsinki.  
firstname.lastname@helsinki.fi

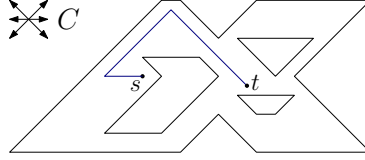


Figure 1: A  $C$ -oriented domain and a minimum-link  $C$ -oriented path in it.

3SUM-hard; this answers a question from the survey [26] and Problem 22 in The Open Problems Project [7].

Our 3SUM-hardness proof suggests that the problem’s complexity stems from allowing the path edges to go in arbitrary directions. This—along with practical considerations—motivates the restricted,  $C$ -oriented setting [1, 11–13, 27, 31, 36] (Fig. 1) in which orientations of path edges come from a fixed set  $C$  of directions. (Abusing notation, we use  $C$  to denote also the cardinality of the set  $C$ .) Adegeest, Overmars and Snoeyink [1] presented two algorithms for finding minimum-link  $C$ -oriented paths in  $C$ -oriented domains – one running in  $O(C^2n \log n)$  time and space, the other in  $O(C^2n \log^2 n)$  time and  $O(C^2n)$  space.

In Section 3 we present an  $O(C^2n \log n)$ -time  $O(Cn)$ -space algorithm, slightly improving on both algorithms from [1].<sup>1</sup> As a by-product, in Section 3.1, we reestablish the optimal time and space bounds claimed in [38] for computing a minimum-link *rectilinear* path amidst rectilinear obstacles. Unlike the earlier papers on the rectilinear case, we use only elementary data structures, which simplifies the algorithm and its analysis. We also show how to find a  $C$ -oriented path in a general domain (Section 3.3.1), give an  $O(Cn \log n)$ -time  $O(n)$ -space 2-approximation algorithm for  $C$ -oriented paths (Section 3), and investigate in what sense  $C$ -oriented paths can approximate minimum-link paths with unrestricted orientation (Section 3.3.3).

All of our algorithms not only find minimum-link paths but also build, within the same time and space bounds, the corresponding link distance maps—exact or approximate. For instance, using our algorithms, one can construct approximate (additive or multiplicative) maps for general minimum-link paths in general domains in subquadratic time and linear space. This is in contrast with the exact link distance maps, which may have quartic complexity [34].

## 2 Paths with unrestricted orientations

The 3SUM-hardness of finding a minimum-link path can be seen easily, as we now observe. Start from an instance of the 3SUM-hard problem GeomBase considered in [8]: Given a set  $S$  of points lying on 3 parallel lines  $l_1, l_2, l_3$ , do there exist 3 points from  $S$  lying on a line  $l \notin \{l_1, l_2, l_3\}$ ? Construct an instance of the minimum-link path problem as follows (Fig. 2, left):  $l_1, l_2, l_3$  become obstacles, and each point  $p \in S$  is a gap punched in the obstacle. The  $s$ - $t$  link distance is 3 if and only if there exist 3 collinear gaps  $p_i, i = 1, 2, 3$ , such that  $p_i \in l_i$ .

We thus obtain:

**Theorem 2.1.** *Determining the link distance, for paths with unrestricted orientations, between two points of a polygonal domain with holes is 3SUM-hard. In particular, it is 3SUM-hard to decide if there exists a 3-link path between two points in a rectilinear domain.*

*Remark 1.* One can decide if the link distance between points  $s$  and  $t$  is 1 in time  $O(n)$  (just test the segment  $st$  for intersection with each edge of the domain). One can test if the  $s$ - $t$  link distance is  $\leq 2$  in time  $O(n \log n)$  (just compute the visibility polygons with respect to  $s$  and  $t$ , in time  $O(n \log n)$ , and test them for intersection, in time  $O(n)$ ). One can test if the  $s$ - $t$

<sup>1</sup>The algorithm was also presented at WADS 2011 [30].



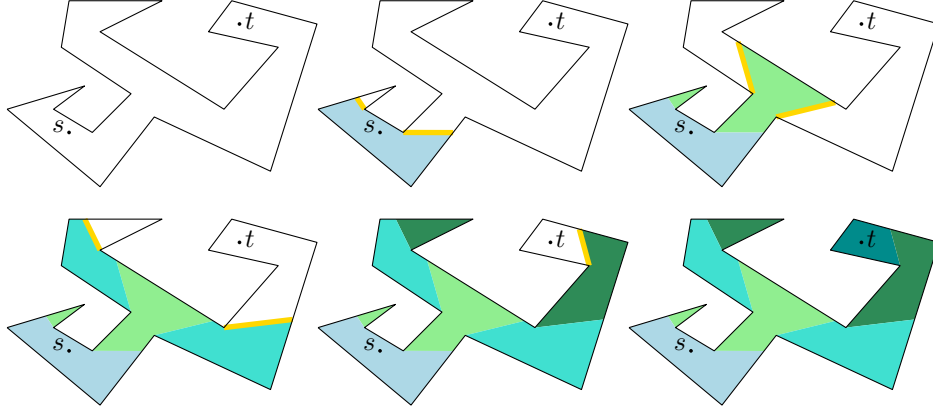


Figure 3: Staged illumination in a simple polygon. The windows are yellow.

opaque; rather, they are “semi-transparent”: we triangulate the simple polygon and do the staged illumination from  $s$ , but, as we go, each time a bridge is illuminated on one of its sides, at the next stage we consider it to be illuminated also on its other side, and continue the staged illumination. Let  $\mathbf{opt}$  denote an optimal path, and, abusing notation, also the number of links in it. In comparison to  $\mathbf{opt}$ , we are delayed by 1 link each time an edge of  $\mathbf{opt}$  crosses a bridge. That is, on every edge of  $\mathbf{opt}$  we have as many additional vertices as there are bridges that the edge crosses. Using a low-stabbing-number tree for the bridging [3], we ensure that each edge of  $\mathbf{opt}$  crosses  $O(\sqrt{h})$  bridges, and thus we obtain an  $O(\sqrt{h})$  multiplicative approximation.

As described above, the illumination takes  $O(nh)$  time, which is quadratic in the worst case, since each of the  $\Omega(n)$  triangles can potentially be discovered by light emanating from  $h$  different bridges (Fig. 5). To address this inefficiency, we declare a triangle opaque as soon as it is lit  $m$  times, for a parameter  $m \leq h$  (Fig. 6). We prove that with the right choice of  $m$  this does not delay the illumination by too much, while decreasing the runtime of the illumination to  $O(nm)$ .

### 2.1.1 The algorithm

We compute a set  $B$  of  $h$  line segments, called *bridges*, such that the *cut polygon*  $P^{\succ} = P \setminus B$  is a (weakly) simple polygon (i.e.,  $B$  bridges the holes to  $P$ ’s outer boundary) and such that any line intersects  $O(\sqrt{h})$  bridges. Section 2.2 details how this can be done in  $O(n \log n + h\sqrt{h} \log h + n\sqrt{h})$  time. We triangulate the cut polygon  $P^{\succ}$ , and do the staged illumination in it, modified as follows:

**Modification M1:** We do not compute the windows exactly. Instead, we consider a triangle to be fully lit even if only part of it is illuminated. This simplifies the algorithm, since the boundary of the illuminated region now consists of edges of triangles. To account for a possible underestimation of the link distance that arises from treating partial illumination as full illumination, we add an additional link to our computed  $s$ - $t$  path inside every lit triangle (Fig. 7). Overall this at most doubles the number of links in the output path in comparison to the stage at which  $t$  is illuminated at the termination of our algorithm.

**Modification M2:** Suppose a triangle having a bridge  $b \in B$  as its side gets illuminated at stage  $k$ . As with the standard staged illumination, at stage  $k$  we do not continue the illumination across  $b$  (because  $b$  is an obstacle, a portion of the boundary of the cut polygon  $P^{\succ}$ ). However, at stage  $k + 1$  we treat  $b$  as one of the windows, and, thereby, continue the illumination on the other side of the bridge.

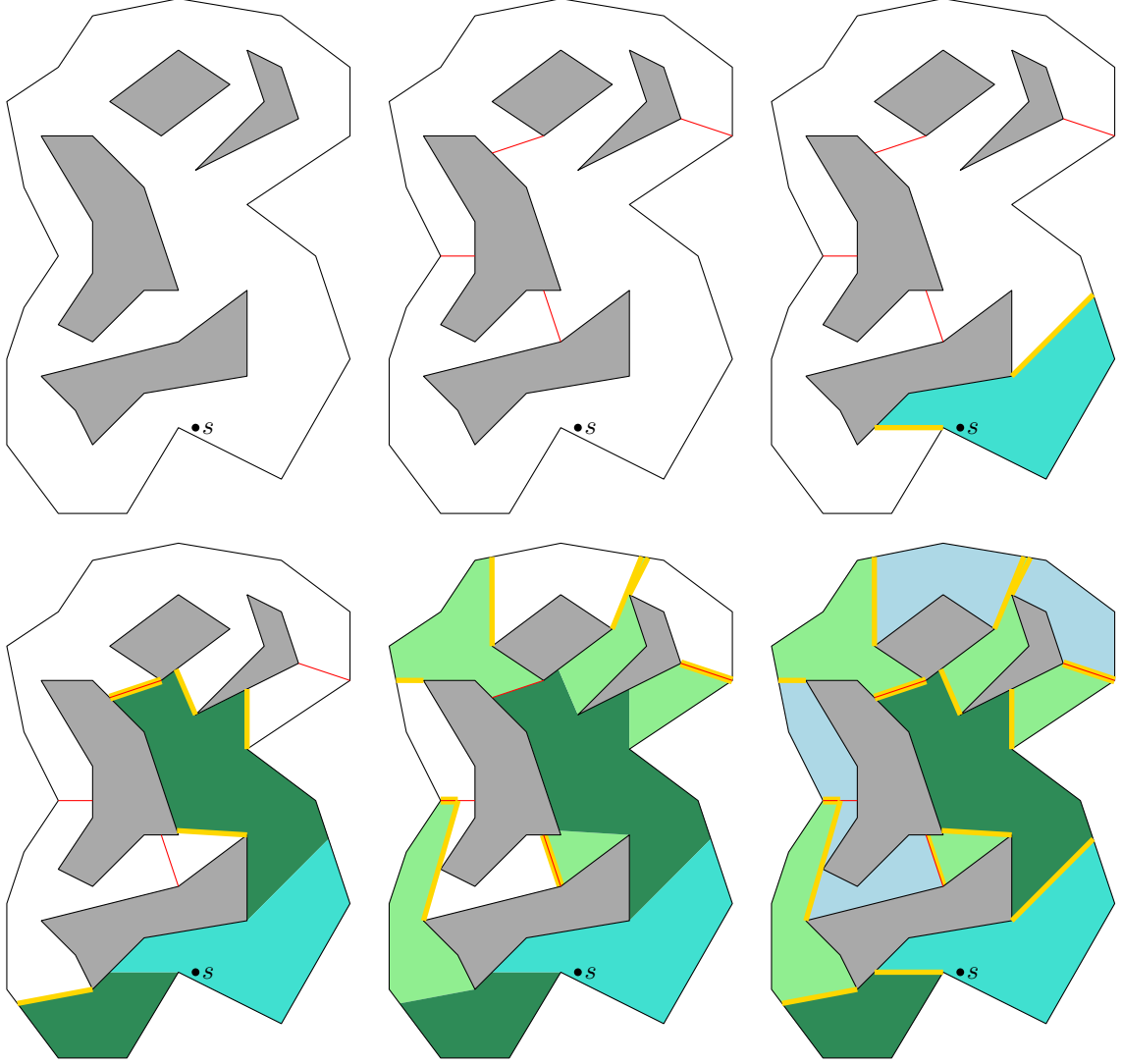


Figure 4: Bridges are red. They block the light, but in the next stage they become windows and emit light on the other side.

The second modification is the main difference between our illumination procedure and the standard one. The difference is best explained by identifying each element in  $B \cup \{s\}$  with a “color” (there are thus  $h + 1$  colors). We will use the convention that anything colored with a color  $b \in B \cup \{s\}$  gets prefix “ $b$ –”; e.g., a window with color  $b$  is a  $b$ -window, etc.

Our illumination starts at  $s$ , and before any bridge is reached by it, all lit triangles are colored with  $s$  (Fig. 8). So far, the process does not differ from the standard illumination (modulo the modification **M1**). In particular, every triangle is  $s$ -lit only once because each  $s$ -window  $w_s$  cuts out a unique dark portion  $P_{w_s} \subset P^{\triangleright^*}$  into which  $w_s$  shines with the  $s$ -light;  $P_{w_s} \cap P_{w'_s} = \emptyset$ ,  $\forall w'_s \neq w_s$ .

Assume now that a bridge  $b$  is reached by the illumination at stage  $k$ . At stage  $k + 1$ ,  $b$  starts shining  $b$ -light on its other side. Since the cut polygon  $P^{\triangleright^*}$  is a simple polygon, each triangle is  $b$ -lit only once. However, nothing prevents a triangle from being seen both from an  $s$ -window and from a  $b$ -window (and, in general, from a window of any other color). Thus, any triangle can be discovered as many times as there are colors ( $h + 1$ ). This means that the overall time of

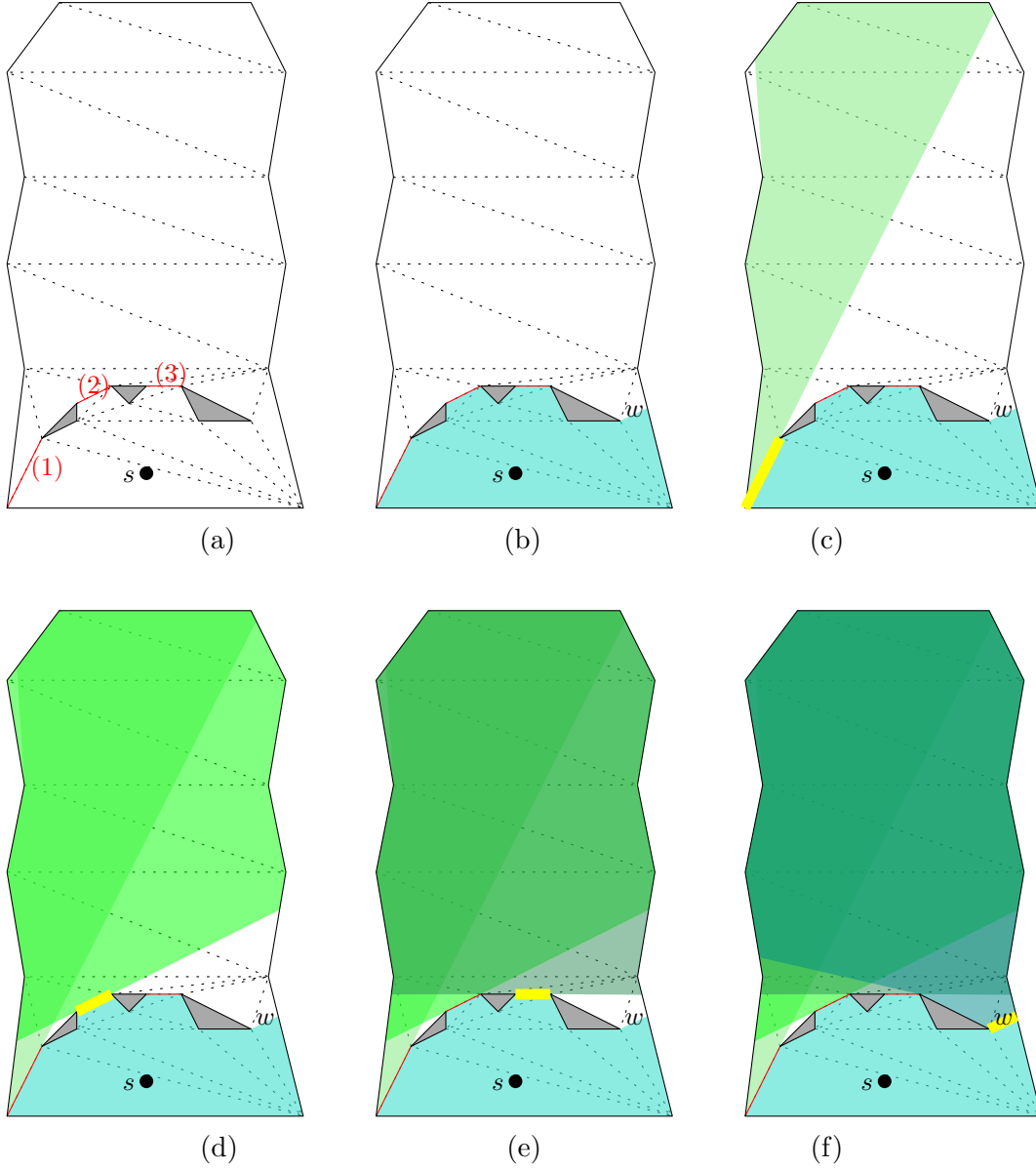


Figure 5: (a) The original domain, the bridges (shown in red) and a triangulation. (b) The region illuminated at the first stage, i.e., the region that is seen from  $s$ .  $w$  is the only window. (c) The region illuminated at the second stage from bridge (1). (d) Regions illuminated at the second stage from bridges (1) and (2). (e) Regions illuminated at the second stage from bridges (1), (2) and (3). (f) Regions illuminated at the second stage from all the bridges and the window  $w$ . There are  $\Omega(n)$  triangles of the triangulation that are illuminated from each of the bridges (and, in addition, from the window  $w$ ).

the illumination is  $O(nh)$ .

Recall that  $\text{opt}$  denotes an optimal path, and also the number of links in the path. Let  $\text{apx}$  denote the  $s$ - $t$  path found by our illumination procedure, and also the number of links in the path (i.e., the stage at which  $t$  is reached).

**Lemma 2.3.**  $\text{apx} = O(\text{opt} \sqrt{h})$ .

*Proof.* Because we use a low-stabbing-number tree for the bridging, any link of  $\text{opt}$  intersects

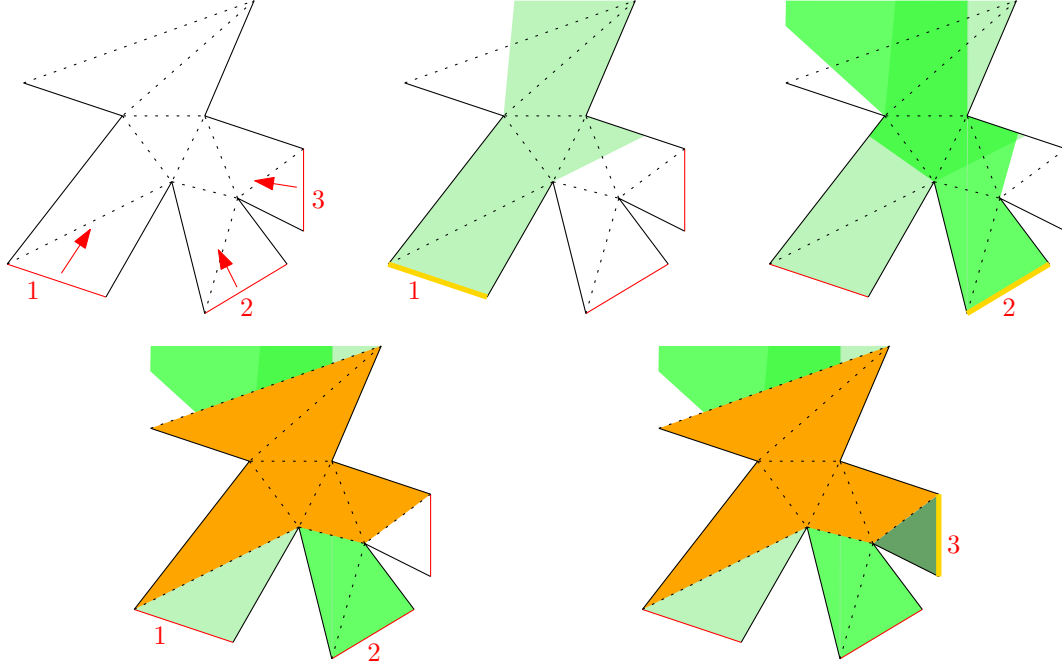


Figure 6: An example using  $m = 2$ . Three windows are shown in red, labeled 1, 2, and 3; arrows indicate the direction in which the light will propagate from the windows. After windows 1 and 2 have propagated the light, some triangles (shown in orange) are 2-lit and are made opaque. Because of this, the light from window 3 lights up only the dark green triangle, but not anything further.

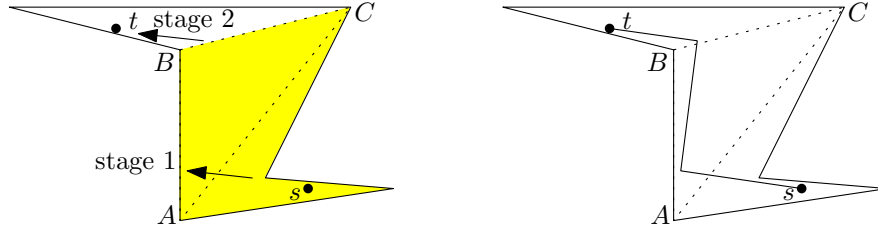


Figure 7: Left: Even though only part of the triangle  $ABC$  is seen from  $s$ , we fully light up the triangle at the first stage. Because of this, at the second stage we shine light from the side  $BC$  and reach  $t$  already at the second stage, even though there exists no 2-link  $s-t$  path. Right: A 3-link path output by our algorithm. The middle link is added to account for the underestimation of the link distance brought by the modification **M1**.

$O(\sqrt{h})$  bridges. Thus, our illumination procedure will spend at most  $O(\sqrt{h})$  stages to propagate light along any one link of **opt**.  $\square$

Since  $h$  can be  $\Theta(n)$ , we do not have a subquadratic algorithm yet. The  $O(nh)$  running time is due to the possibility that a triangle can be lit by more than one color. To speed up the algorithm, we further modify the illumination procedure as follows:

**Modification M3:** Fix a number  $m = o(h)$ . At any stage do the illumination color-by-color. Declare a triangle as an obstacle as soon as it has been reached by  $m$  colors. That is, after being lit by  $m$  colors, any triangle acts as an obstacle for light of any other color that tries to shine through it: it simply blocks the light.

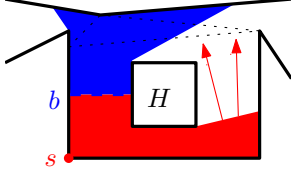


Figure 8: Point  $s$  and the region illuminated at stage 1 are shown in red. Red arrows are some red illumination rays at stage 2. Bridge  $b$  (dashed blue) bridges a hole  $H$  to the outer boundary. Crossing the bridge delays the illumination by one stage, and changes the color of the light to that of the bridge. Even though in the original domain  $P$ , the triangle  $T$  (dotted) is seen directly from  $s$ , the triangle will be lit only at stage 2.  $T$  is lit by both red and blue (after which the colors can be merged).

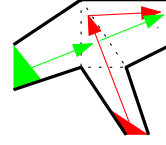


Figure 9:  $\text{apx}$  has a green link, and green light is blocked by red at stage  $k$  by a triangle  $T$  (dotted): this means that  $T$  has been reached with red after  $k$  links (just as with green). Anything reachable with green via  $T$  can be reached also with red, at the expense of a possible extra turn inside  $T$ .

Let  $\text{apx}_m$  denote the  $s$ - $t$  path found by the staged illumination with modifications **M1-M3**. With any triangle being discovered at most  $m$  times, the running time to find  $\text{apx}_m$  goes down to  $O(nm)$ . To bound the number of links in the path (denote it by  $\text{apx}_m$  too), consider the path  $\text{apx}$ . Its links are colored according to the illumination with modifications **M1,M2**. Suppose that one of the links is green. If green color is blocked due to the modification **M3** (say, at stage  $k$  by a triangle  $T$ ), we know that  $T$  has been lit at stage  $k$  by some other (at least  $m$ ) colors. Any of these colors propagates into the region into which green would have propagated had it not been blocked (Fig. 9). Thus, we may have an extra turn in  $\text{apx}_m$  for each stage when a color of  $\text{apx}$  gets blocked. Denoting by  $X$  the overall number of stages at which blocking happens during execution of our algorithm, we have

**Lemma 2.4.**  $\text{apx}_m \leq \text{apx} + X$ .

We emphasize that the difference  $\text{apx}_m - \text{apx}$  depends on the number of *stages* at which blocking happens, not on the total number of blockings encountered throughout the algorithm—blockings that happen outside links of  $\text{apx}$  do not contribute to the difference. Also, note that the path  $\text{apx}$  and the colors of its links enter only the *analysis* for  $\text{apx}_m$ ; in the algorithm for  $\text{apx}_m$  we, of course, do not compute  $\text{apx}$  and do not (have to) know its links' colors.

What remains is to bound  $X$ . To do so we relate double-lighting to color-merging via the following observation: if a triangle  $T$  can be lit both by a red color  $r$  and a green color  $g$  (i.e., if  $T$  is seen from an  $r$ -window and a  $g$ -window) at stage  $k$ , then no triangle  $T' \neq T$  may ever be  $r$ -lit through one side and be  $g$ -lit through another side after stage  $k$ . That is, if  $T'$  is double-lit, the two colors  $r, g$  must arrive through the same side of  $T'$ . Hence the two colors effectively can be merged into a single color. To see why this is the case, recall that any color lights up triangles following the dual of the triangulation of the cut polygon  $P^{\times s}$  (which is a simple polygon). If red and green arrive at  $T'$  via different sides, then there are two distinct  $T$ - $T'$  paths (one followed by red light, one followed by green) in the dual—a contradiction (Fig. 10).

*Remark 2.* In terms of the dark portions  $P_w \subset P^{\times s}$  cut out by windows, the merging formally means that the collection of sets  $P_w$ , for  $w$  ranging over all windows, is a laminar family (if two sets are not disjoint, then one is contained in the other). This is not as strong as a family of pairwise-disjoint sets (as is the case for staged illumination in simple polygons) but is good enough to ensure that each triangle effectively is discovered only once.



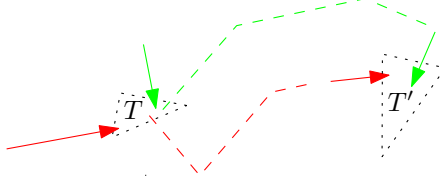


Figure 10:  $T$  was  $g$ -lit and  $r$ -lit at some stage. If later  $T'$  is  $g$ -lit and  $r$ -lit through different sides, then there are two distinct  $T$ - $T'$  paths in the dual of the triangulation of the simple polygon  $P^{\times}$ .

**Lemma 2.5.**  $X \leq h/m$ .

*Proof.* Recall that we are considering blocking that occurs along a single path. There are  $h + 1$  colors to start with. Each time a blocking occurs, a triangle has been seen from windows of  $m$  colors plus a window of the blocked color.  $\square$

We emphasize that the blocking is algorithmic (i.e., we do block during the illumination) while the merging is used only in the analysis. (Of course, we could maintain the colors in a union-find data structure and merge them, but we do not need to do it: even if we propagate the colors unmodified, the merged colors will act as a single color exactly in the sense that no triangle will be discovered by more than one of them.)

From Lemmas 2.3, 2.4, 2.5 we have:

**Proposition 2.6.** *If  $B$  is given, then for any  $m$ , in  $O(nm)$  time one can find a path with  $O(\text{opt} \sqrt{h}) + h/m$  links.*

Choosing  $m = \Theta(\sqrt{h})$ , we obtain the main result of this section:

**Theorem 2.7.** *An  $O(\sqrt{h})$ -approximate minimum-link path can be found in  $O(n\sqrt{h} + n \log n + h\sqrt{h} \log h)$  time.*

Our algorithm labels the triangles with approximate link distance to  $s$  and thus also builds a linear-size approximate link distance map in subquadratic time (we recall that the exact map can have  $\Theta(n^4)$  complexity [34]).

*Remark 3.* With the cutting of  $P$  into  $P^{\times}$  and propagating light on the other sides of bridges, our approach resembles, in some sense, working in the universal covering space of  $P$  [13].

**$O(\sqrt{h} \log h)$ -approximation with  $m = 1$**

The bottleneck in the algorithm of Theorem 2.7 is computing the bridges  $B$ . If a more efficient solution is designed for the bridging, it will be of interest to do the illumination faster, too. There may also be other reasons to speed up the illumination: e.g., note that the bridges are good irrespective of the choice of  $s$  and  $t$ ; thus, one may use the same  $B$  when finding minimum-link paths between different pairs  $s, t$ . (Of course, it would be more interesting to design a special data structure to do 2-point approximate link distance queries efficiently; this is however beyond the scope of the current paper.)

Our illumination is fastest with  $m = 1$  (i.e., when any triangle becomes an obstacle and blocks light as soon as it is lit by *any* color). However, with  $m = 1$  Proposition 2.6 does not give a multiplicative approximation guarantee (because  $\text{opt}$  can be  $o(\sqrt{h})$ ). We now show that with yet another modification, one can use  $m = 1$  and guarantee an  $O(\sqrt{h} \log h)$  approximation factor while only adding  $O(h \log h)$  to the time of the illumination.

Specifically, our last modification is as follows:

**Modification M4:** Identify colors with (not necessarily distinct) integers, and at each stage do the illumination color-by-color in decreasing order (with ties broken arbitrary). When a color first becomes active, it is identified with 1. When a color  $g$  gets blocked by a color  $r$ , merge the colors into a single color  $r + g$  (i.e., any color equals to the number of the colors merged into it).

Note that with this modification, we do the color merging algorithmically. To do it efficiently, we store the colors, in the order in which we illuminate from them, in a binary search tree. The tree is updated each time a merge occurs. There can be at most  $h$  merges, so the total time spent in updating the tree is  $O(h \log h)$ .

Let  $\text{apx}_1$  denote the number of links in the  $s$ - $t$  path found by the staged illumination with modifications **M1-M4**.

**Lemma 2.8.**  $\text{apx}_1 = O(\text{apx} \cdot \log h)$ .

*Proof.* Consider a link  $ab$  of the path  $\text{apx}$  (the path found with modifications **M1-M2**). Suppose that when we do the illumination with all modifications **M1-M4**, the triangle containing  $a$  is lit with a green color, identified with integer  $g$ . If green light fails to reach  $b$ , it must be blocked by some other color, say a red color, identified with integer  $r$ . This means that  $r \geq g$ . Since  $r$  and  $g$  bump into each other, they get merged into a single color  $r + g \geq 2g$ . That is, whenever propagation is delayed at some stage, the color number at least doubles. Thus, after  $O(\log h)$  such delays, the illuminating color is at least  $h/2 + 1$ , which means we are illuminating with the highest possible color and no color remains to block light from reaching  $b$ .  $\square$

We thus obtain the following variation of Proposition 2.6:

**Proposition 2.9.** *If  $B$  is given, in  $O(n + h \log h)$  time one can find a path with  $O(\text{opt} \sqrt{h} \log h)$  links.*

*Remark 4.* The improvement brought by the modification **M4** may look surprising: the original illumination description did not specify in what order the illumination is done using different colors; that is, the order was arbitrary. With the modification, the order is still quite arbitrary because the tie breaking is arbitrary (and ties may happen often). The reason the modification helps is that the illumination is done in an arbitrary, but nevertheless systematic and stage-to-stage consistent way.

## 2.2 Computing the bridges

We describe the algorithm for computing bridges, following the exposition from [14, Section 7]. The technique, called “well-known” already in [14], dates back to [3] and is based on using *low-stabbing-number* spanning trees for point sets. To our knowledge, such trees were first described in [35]; see also [24, Section 5] and [2, 4, 20].

**Lemma 2.10** ([35]). *Let  $R$  be a set of  $h$  points in the plane. In  $O(h\sqrt{h} \log h)$  time one can compute a (non-self-crossing) spanning tree  $T$  of  $R$  such that any line in the plane intersects  $O(\sqrt{h})$  edges of  $T$ .*

Recall that the goal of the bridging is to compute a set  $B$  of  $h$  line segments (the bridges) to connect up the holes and the boundary of  $P$  into a (weakly) simple polygon.

We surround  $P$  with a large bounding box  $\mathcal{B}$  and compute, in time  $O(n \log n)$ , a full triangulation within  $\mathcal{B}$ , of  $P$  and its holes, as well as the portion of  $\mathcal{B}$  outside the outer boundary of  $P$  (we think of this region as another “hole”). Refer to Fig. 11, top left.

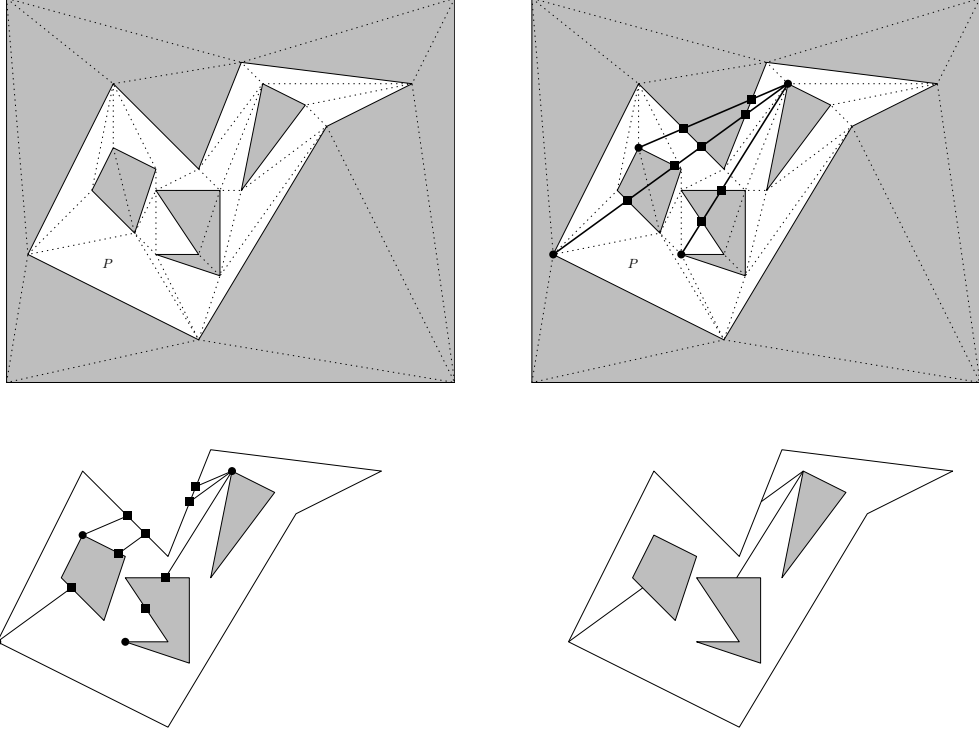


Figure 11: Top left: The triangulations (dotted). Top right: The spanning tree  $T$  (thick edges) of the representative vertices (circles), and the intersections (squares). Bottom left: Each segment connects two different holes. Bottom right: The computed bridges.

Next, pick one representative vertex from each hole ( $h + 1$  representatives altogether) and build the low-stabbing-number spanning tree  $T$  of the representatives (the tree has  $h$  edges). The edges of  $T$  do not cross each other, but they may intersect boundaries of holes. To discover such intersection points, take each edge of  $T$  and trace it using the triangulation. By Lemma 2.10, each diagonal crosses  $O(\sqrt{h})$  edges of  $T$ , so that the total number of triangles crossed by all edges of  $T$  is  $O(n\sqrt{h})$ . Thus, in total  $O(n\sqrt{h})$  time we can discover all of the intersection points. Refer to Fig. 11, top right.

The intersection points subdivide the edges of  $T$  into a total of  $O(n\sqrt{h})$  smaller segments. Each segment either connects two boundary points of the same hole (the segment may lie fully inside the hole or fully outside it), or connects two different holes. Remove the former segments. Now each segment connects two different holes. Refer to Fig. 11, bottom left.

Examine the segments one by one, in an arbitrary order, and merge all holes into one (using a union-find data structure). That is, for each segment in turn, check whether it connects the boundary points of the same hole or of different holes. In the former case, remove the segment. In the latter case, keep the segment as one of the bridges in  $B$  and merge the holes that it connects into one hole. Since we start with  $h + 1$  holes,  $B$  will contain  $h$  bridges when all holes are merged. Refer to Fig. 11, bottom right.

Finally, to obtain the (weakly) simple cut polygon  $P^{\succ} = P \setminus B$ , slice  $P$  along the bridges (each bridge becomes two edges of the cut polygon  $P^{\succ}$ ). It follows from the above that the cut polygon can be built in  $O(n \log n + h\sqrt{h} \log h + n\sqrt{h})$  time and that it has the desired property – any line in the plane intersects any bridge  $O(\sqrt{h})$  times.

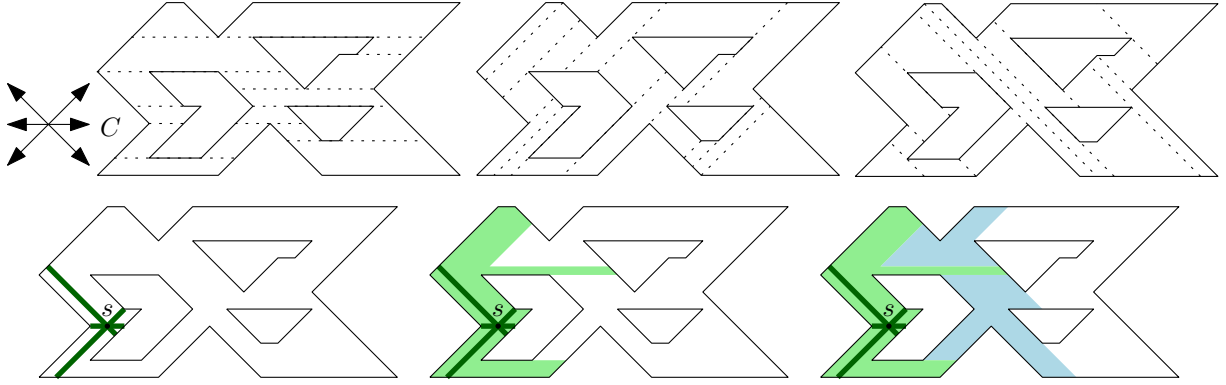


Figure 12: Top: The trapezoidations. Bottom: The regions reachable with 1, 2 and 3 links.

### 3 $C$ -oriented paths

In this section  $P$  is a  $C$ -oriented domain. To find a minimum-link  $C$ -oriented  $s$ - $t$  path (and to construct a  $C$ -oriented link distance map from  $s$ ) in  $P$ , we follow the general approach of [1]: Build  $C$  trapezoidations of  $P$ , where the trapezoids in the trapezoidation  $c \in C$  are obtained by extending maximal free  $c$ -oriented segments through vertices of  $P$  (Fig. 12). Label the trapezoids with their link distance from  $s$ . This way an  $O(Cn)$ -space structure is created that answers link distance queries in  $O(C \log n)$  time. The labeling proceeds in  $n$  steps, with label- $k$  trapezoids receiving their label at step  $k$ . Any label- $k$  trapezoid must be intersected by a label- $(k - 1)$  trapezoid of a different orientation. Hence, step  $k$  boils down to detecting all unlabeled trapezoids intersected by label- $(k - 1)$  trapezoids. In terms of the *intersection graph* of trapezoids (which has nodes corresponding to trapezoids and edges corresponding to intersecting pairs of trapezoids), the labeling is done by Breadth First Search (BFS) starting from the  $C$ -oriented segments through  $s$ .

The idea of performing an efficient BFS in the intersection graph without explicitly building the (potentially quadratic-size) graph dates back to the work on minimum-link *rectilinear* paths [5, 6, 17, 18, 22, 23, 28, 32, 37–39]. As noted already in [5, 37], Imai and Asano’s [17, 18] data structure allows one to do the BFS—and hence to find a minimum-link rectilinear path—in  $O(n \log n)$  time and space. Still, it was not until the 1991 work of Das and Narasimhan [5] (and the lesser known work of Sato, Sakanaka and Ohtsuki [32]) that an optimal,  $O(n \log n)$ -time  $O(n)$ -space BFS implementation was developed (see also Appendix A). In the implementation, one step of the BFS is reduced to a pair of sweeps—the UpSweep and the DownSweep.

The data structures employed in [5, 32] are simpler than the (much more general-purpose) structure of Imai and Asano [17, 18], but they are still more complicated than one would hope for the basic problem of finding rectilinear paths amidst rectilinear obstacles.<sup>2</sup> In Section 3.1 we present a simplified implementation of the BFS step in the intersection graph. Our modification does not affect the asymptotic time and space optimality of the algorithm. The crux of the simplification is the use of a single tree for storing the intersection of the sweepline with the trapezoids that were lit at the previous step.

Another minor modification in our algorithm comes from performing only one (upward) sweep at any step of the BFS. The sweep starts from what we call the “pot” trapezoids—those into

<sup>2</sup>The conference paper [5] admits omitting the details in several places; to our knowledge, no fuller version exists. In particular, we were not quite able to parse the description in [5], in which the light propagates from “portions” of “portions” of “outermost” segments discovered at the previous step: according to the 5th paragraph on p. 265, at any step light is directed from “portions” of the fronts, but the fronts themselves seem to be “portions” of the “outermost” segments lit at the previous stage (4th paragraph).

which the different-orientation trapezoids lit at the previous step are “planted”. The planting is nothing but a means to initialize the sweep (without the planting, it is not clear how to efficiently discover even a single edge in the intersection graph). While for the rectilinear case our modification saves only a factor of 2 in running time, it serves as the basis of our improvements for the general case of  $C$ -oriented paths with  $C > 2$  (Section 3.2).

*Remark 5.* In Section 3.3.2 we also use the original algorithm of [5]—with both the UpSweep and DownSweep—to give a more time-space efficient 2-approximation algorithm for minimum-link  $C$ -oriented paths.

### 3.1 Rectilinear paths in rectilinear domains

In this subsection  $P$  is a rectilinear domain, and we want to build the rectilinear link distance map from  $s$ . For simplicity of exposition, we make a non-degeneracy assumption that no two edges of  $P$  are supported by the same line; it is straightforward to handle the degenerate cases. As in [5], we start by forming 2 decompositions of  $P$ : one by extending maximal free horizontal segments supported by horizontal edges of  $P$ , and the other by extending vertical segments. The horizontal decomposition is stored in a linked structure that links any cell to its upper neighbors. The vertical decomposition is stored analogously. Both decompositions are rectangular; however, to connect better to our techniques for general  $C$ -oriented paths (Section 3.2), we call the rectangles in the horizontal decomposition *trapezoids*, and those in the vertical decomposition we call *parallelograms*. The horizontal edges of a trapezoid are its *bases*, and the vertical edges are its *sides*. For a parallelogram, on the contrary, its vertical edges are bases, and horizontal edges are the sides. Any side is thus a subset of an edge of  $P$ .

Our goal is to label the cells of the decompositions (i.e., trapezoids and parallelograms) with link distance from  $s$ . If a trapezoid (resp., parallelogram) has label  $k$  then every point inside the trapezoid (resp., parallelogram) can be reached with a  $k$ -link path whose last link is horizontal (resp., vertical), and some point inside the trapezoid cannot be reached with fewer than  $k$  links. As discussed above, the labeling amounts to BFS in the trapezoids–parallelograms intersection graph, starting from maximal free horizontal and vertical segments through  $s$  (which are labeled with 1). Step  $k$  of the BFS is as follows: given a subset  $S_*^{k-1}$  of parallelograms (those labeled  $k-1$ ), find all unlabeled trapezoids intersected by parallelograms in  $S_*^{k-1}$ . Denote the set of the sought trapezoids by  $S^k$ . (Strictly speaking, this is only half of the BFS step; the other step is analogous, with the roles of trapezoids and parallelograms reversed.) The idea is to discover the trapezoids from  $S^k$ , in the order of increasing  $y$ -coordinate of the lower bases, by sweeping a horizontal line upwards.

The remaining question is how to initialize and maintain the sweep event queue. We do not want to insert *all* unlabeled trapezoids into the queue (as it could lead to linear-size queues on a linear number of the BFS steps). On the other hand, the trapezoids from  $S^k$  *must* appear in the step- $k$  queue at some point. Of course, ideally, *only* the trapezoids from  $S^k$  should ever appear in the queue; we, however, did not see how to enforce this.

Instead, we insert a *superset* of  $S^k$  into the queue. Every trapezoid  $T$  inserted in the queue at step  $k$  is either itself intersected by a parallelogram from  $S_*^{k-1}$ , or it has a lower neighbor intersected by a parallelogram from  $S_*^{k-1}$  (or both). In the former case, the label of  $T$  may not be smaller than  $k-2$  and may not be larger than  $k$ . In the latter case, the label of  $T$  is not larger than  $k+2$  and not smaller than  $k-4$  (Fig. 13, left). That is, the label of any trapezoid queued at step  $k$  belongs to the interval  $[k-4, k+2]$ . Viewed differently, this means that any trapezoid is processed during at most 7 steps (7 is not a tight bound, but all we need is that it is a constant). This gives an  $O(n)$  bound on the total number of events over all  $n$  BFS steps. Since there are  $O(n)$  trapezoids, the claimed  $O(n \log n)$  time and  $O(n)$  space bounds for the

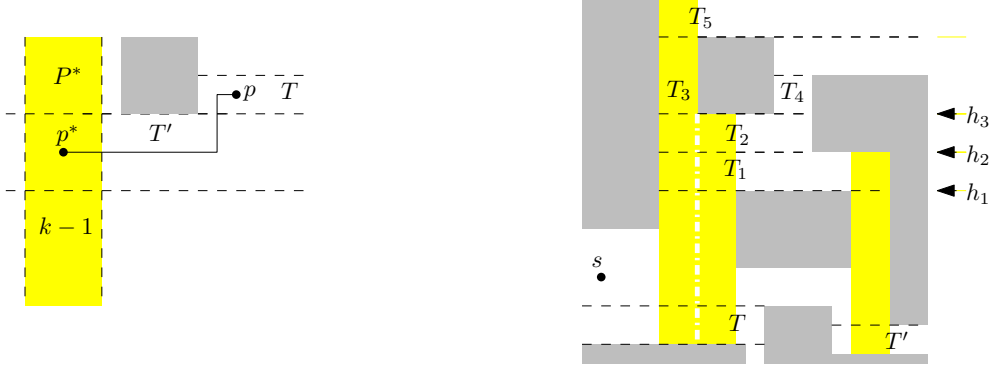


Figure 13: Left: If  $T$  has a lower neighbor  $T'$  intersected by a parallelogram  $P^* \in S_*^{k-1}$ , then from any point  $p \in T$  there exists a 3-link path to some point  $p^* \in P^*$ . Thus, the label of  $T$  is within 3 of  $k - 1$ , the label of  $P^*$ . Right: Three parallelograms from  $S_*^2$  are shown yellow. The two left parallelograms are planted into  $T$ , the rightmost parallelogram is planted into  $T'$ . Some of the events are as follows: At  $h_1$ , the trapezoid  $T_1$  is the event. Since its lower base overlaps with intervals in the sweepline status,  $T_2$  is inserted into the event queue. Since  $T_1$  is dark, it gets its label 3. At  $h_2$ , the first event is the update of the sweepline status. The next event at  $h_2$  is  $T_2$ :  $T_3, T_4$  are inserted into the queue and  $T_2$  is labeled. At  $h_3$ , first the sweepline status is updated; then,  $T_3$  is processed: since it is intersected by the parallelograms from  $S_*^2$  (as witnessed by the sweepline status intervals),  $T_5$  enters the queue and  $T_3$  is labeled. Finally,  $T_4$  is processed: its lower base does not overlap intervals from the status, so it remains unlabeled.

algorithm follow once we show how to process an event in  $O(\log n)$  time.

The correctness of the sweep follows from the standard invariant: all trapezoids from  $S_k$  whose lower bases are below the sweepline are labeled. We now give the details of the sweep: its initialization, sweepline status and the events.

**Planting** We say that a parallelogram  $P^* \in S_*^{k-1}$  is *planted* into a trapezoid  $T$  if the lower side of  $P^*$  overlaps with the lower base of  $T$  (Fig. 13, right). We say also that  $T$  is the *pot* of  $P^*$ . We initialize the event queue by inserting all pots into it. The crucial observation is that each parallelogram is planted into exactly one pot (even though a pot can have many parallelograms planted side-by-side into it). Moreover, it is straightforward to augment the data structures storing the decompositions so that for any parallelogram its pot can be determined in  $O(1)$  time (store with each parallelogram the horizontal edge of  $P$  that supports its lower side, and store with each edge of  $P$  the trapezoid whose lower base supports the edge).

**Parallelogram events** The sweepline status is the intersection of the sweepline with the (interiors of) parallelograms from  $S_*^{k-1}$ . The status thus is a set of disjoint (open) intervals. The status changes at *parallelogram events* when the sweepline reaches sides of parallelograms. If the event is an upper side, the side is removed from the status. If the event is a lower side, it is added to the status (the parts of intervals already in the status overlapping with the newly added interval are removed). The total number of parallelogram events over all BFS steps is  $O(n)$  since each label- $(k - 1)$  parallelogram contributes 2 events at step  $k$ . Because the intervals in the status are disjoint, we can keep them in any ordered structure, e.g., a balanced binary search tree indexed by left endpoints of the intervals (the more complicated data structures in [5] were utilized to maintain “fronts” composed of merging and splitting “windows”). Clearly, the tree handles any of the following three operations in amortized  $O(\log n)$  time: (1) adding an

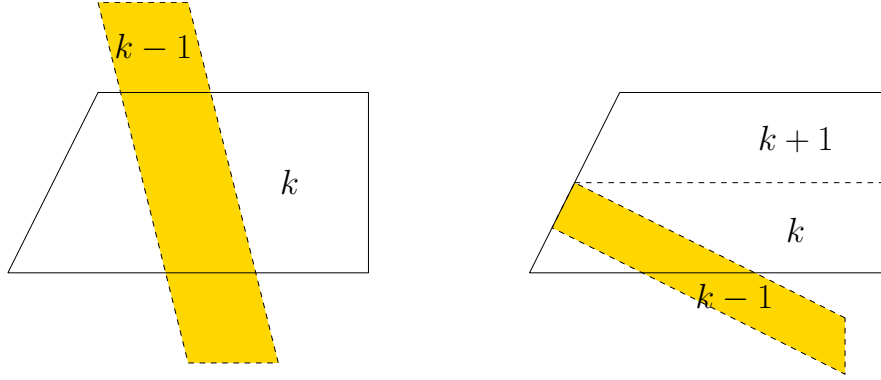


Figure 14: Left: A trapezoid gets fully labeled. Right: A trapezoid is labeled only partially at step  $k$ .

interval (we do not merge the intervals into superintervals because for backtracking we need to know from which parallelogram a trapezoid was lit), (2) removing part of an interval that hits an obstacle edge, and (3) checking whether any of the intervals overlaps with a given query interval. In the last operation, the query interval is a trapezoid lower base. We need it for the trapezoid events, described next.

**Trapezoid events** The main events in the sweep are *trapezoid events* that occur when the sweepline reaches a lower base of a trapezoid (some of the trapezoid events happen simultaneously with parallelogram events; in this case parallelogram events take priority). Suppose that a trapezoid  $T$  is the event. We check whether the lower base of  $T$  is intersected by the intervals in the sweepline status. If yes, we insert all (at most 2, due to non-degeneracy) upper neighbors of  $T$  into the event queue (recall that to facilitate the insertion, the upper neighbors are linked from  $T$ ). In addition, if  $T$  is unlabeled, we label it with  $k$ . Refer to Fig. 13.

### 3.2 $C$ -oriented paths in $C$ -oriented domains

Let  $P$  be a  $C$ -oriented domain. Our goal is to build the  $C$ -oriented link distance map from  $s$ . As noted in [1], the efficient methods developed to perform BFS in the trapezoids intersection graph for the rectilinear version do not extend to  $C$ -oriented paths when  $C > 2$ . Let us look closely at why this is the case. One reason is that for  $C > 2$  some trapezoids may get labeled only partially during a BFS step (Fig. 14). This complicates the BFS because the intersection graph changes from step to step, and, in the final link distance map, trapezoids may get split into subtrapezoids. The partial labeling and splitting are due to the possibility that two different-orientation trapezoids do not “straddle” each other; instead they both may be “flush” with an obstacle edge whose orientation is different from the orientations of both trapezoids (this was not the case in the rectilinear version, since there were only 2 orientations). However, such a flush intersection can be read off easily from lists of incident trapezoids stored with each edge of  $P$ . Thus, discovering partially labeled trapezoids becomes the easy part of the algorithm. For the remaining part we do a sweep for each pair of orientations in  $C$ .

Specifically, after the partially labeled trapezoids are processed, we are left with discovering unlabeled trapezoids “fully straddled” by trapezoids labeled at the previous step. As with the rectilinear case, it is the straddling that leads to a superlinear-size intersection graph and makes a subquadratic algorithm less trivial. It is tempting to reuse here the sweeping techniques developed for the rectilinear version. The stumbling block, though, is choosing the direction of the sweep. Indeed, no matter in which direction the sweep proceeds, the intersection of

the sweepline with a trapezoid does not change only at discrete “events”: while the sweepline intersects a non-parallel side of the trapezoid, the intersection changes continuously. The good news is that this is the *only* reason for a continuous change of the intersection. This prompts us to get rid of the non-parallel sides of the trapezoids by clipping them into parallelograms, with the new sides parallel to the sweepline. After the clipping (and planting the parallelograms appropriately) is done, we are able to reuse the rectilinear-case machinery and finish the BFS step with  $C(C - 1)$  sweeps—one per (ordered) pair of orientations. Overall we obtain an  $O(C^2 n \log n)$ -time  $O(Cn)$ -space algorithm.

The main difference between our algorithm and that of [1] is the separate treatment of flush and straddling trapezoids. This allows us to use only elementary data structures and improve the time-space bounds to  $O(C^2 n \log n)$  and  $O(Cn)$ . We now describe the details.

## Definitions and preliminary observations

We start by defining the notation and recalling some results from [1]. Any trapezoid  $T$  has two opposite edges belonging to the boundary of  $P$ ; these edges are *sides* of  $T$ . The other two edges are  $T$ ’s *bases*; the bases are parallel segments whose orientation belongs to  $C$ . For an orientation  $c \in C$ , a *c-segment* is a segment with orientation  $c$ . A *c-trapezoid*  $T$  has  $c$ -segments as bases. A *subtrapezoid* of  $T$  is a  $c$ -trapezoid cut out from  $T$  by one or two  $c$ -segments.

***c*-distance** A *c-path* is a path (starting from  $s$ ) whose last link is a  $c$ -segment. A point  $p \in P$  is at *c-distance*  $k$  from  $s$  if  $p$  can be reached by a  $k$ -link  $c$ -path (but not by a  $(k - 1)$ -link  $c$ -path).

**The output: *c*-maps** The *c-distance* equivalence decomposition of  $P$  (the *c-map*) is the partition of  $P$  into  $c$ -trapezoids such that the *c-distance* to any point within a cell is the same. If the *c-distance* to points in a  $c$ -trapezoid of the *c-map* is  $k$ , then the  $c$ -trapezoid has *label*  $k$ . Using the illumination analogy we also say that the trapezoid is *lit* at step  $k$ . Unlit trapezoids are *dark*. Denote the set of  $c$ -trapezoids lit at step  $k$  by  $S_c^k$ .

**Convention: *c* is horizontal and implicit** In our algorithm, finding  $S_c^k$  is completely identical to (and independent from) finding  $S_{c^*}^k$  for any  $c^* \in C \setminus c$ . In what follows we focus on finding  $S_c^k$ . Where it creates no confusion, we omit the subscript  $c$  and the prefix “ $c$ ”; e.g., “path to trapezoid in  $S^{k-1}$ ” means “ $c$ -path to  $c$ -trapezoid in  $S_c^{k-1}$ ”, etc. We let  $C^*$  denote  $C \setminus c$ , and use  $c^*$  for a generic orientation from  $C^*$ . Assume, without loss of generality, that  $c$  is horizontal.

***c*-map via trapezoidation refinement** Denote by  $D^k$  the “at-most- $k$ -links map”, i.e., the trapezoidation whose trapezoids are of  $k + 1$  types—dark trapezoids, and trapezoids lit at steps  $1, \dots, k$ . All trapezoids in  $D^k$  are maximal — the dark trapezoids are maximal in the sense that each is a maximal trapezoid every point of which has distance  $k + 1$  or larger, and the lit trapezoids are the same as they are in the final map (i.e., trapezoids lit at step  $k$  are exactly  $S^k$ ). By definition, lit trapezoids from  $D^{k-1}$  remain the same in  $D^k$ ; in particular,  $D^n$  is the *c-map*. Let  $T'$  be a dark trapezoid from  $D^{k-1}$ . The crucial (albeit obvious) observation about minimum-link paths is (cf. [1, 5, 33]):

**Observation 3.1.** *There exists a  $k$ -link path to a point  $p \in T'$  if and only if there exists a  $(k - 1)$ -link  $c^*$ -path  $\pi^*$  to some point  $q \in T'$  that has the same  $y$ -coordinate as  $p$ .*

If  $\pi^*$  enters  $T'$  through its lower base, then any point whose  $y$ -coordinate is smaller than the  $y$ -coordinate of  $p$  is also reachable by a  $k$ -link path. Thus, the set of points of  $T'$  reachable by  $k$ -link paths entering  $T'$  through its lower (resp., upper) base is a subtrapezoid,  $T'_l$  (resp.,



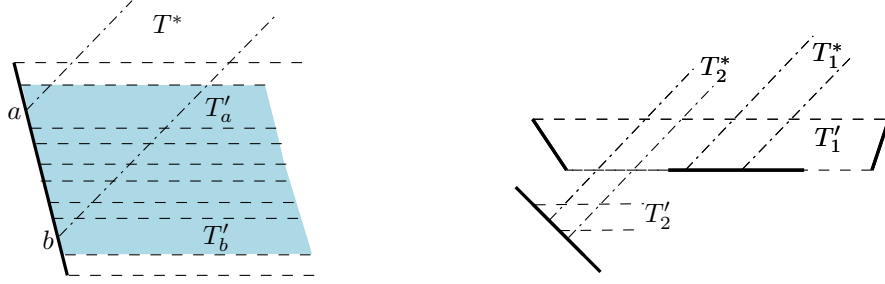


Figure 15: Intersection types. Left:  $T^*$  is flush with the shaded trapezoids.  $T'_b$  will be split in  $D^k$  unless more of it is lit by another trapezoid.  $T'_a$  will be the pot for the parallelogram cut out of  $T^*$  by the  $c$ -segment through  $a$ . Right:  $T_1^*, T_2^*$  straddle  $T'_1$ . The parallelogram cut out of  $T_1^*$  is planted into  $T'_1$ . The pot for the parallelogram cut out of  $T_2^*$  is a trapezoid  $T'_2$  flush with  $T_2^*$ .

$T'_u$ ), of  $T'$  (cf. [1, Fig. 5]). If  $T'_l \cup T'_u = T'$ , we say that  $T'$  is *fully* lit; otherwise, it is *partially* lit. Note that in the latter case the subtrapezoid  $T' \setminus (T'_l \cup T'_u)$  will be fully lit in  $D^{k+1}$ , since orientations of edges of  $P$  belong to  $C$ . Thus,  $D^k$  is a refinement of  $D^{k-1}$ , and overall we have:

**Observation 3.2.** *No trapezoid from  $D^0$  is split into more than 3 trapezoids in  $D^n$ .*

The trapezoidation  $D^0$  is just the trapezoidal decomposition of  $P$  with maximal free segments supported by vertices of  $P$ . Because  $D^0$  is linear-size, by Observation 3.2 so is  $D^n$ , the  $c$ -map. To answer a link distance query from a point  $q \in P$  to  $s$ , it is sufficient to locate  $q$  in each of the  $c$ -maps and choose one in which the trapezoid of  $p$  has the smallest label.

The above discussion reestablishes the space and query time results of [1]: a size- $O(Cn)$  structure can be built to answer link distance queries in  $O(C \log n)$  time. Two approaches were presented in [1] to construct the  $c$ -maps: one using  $O(C^2 n \log n)$  time and space, the other using  $O(C^2 n \log^2 n)$  time and  $O(C^2 n)$  space. We construct the  $c$ -maps in  $O(C^2 n \log n)$  time and  $O(Cn)$  space. Our general approach is still the same as in [1]: at step  $k = 1, \dots, n$ , starting from  $D^{k-1}$ , build  $D^k$  by identifying  $S^k$ —the trapezoids lit at step  $k$ . We find the lit trapezoids differently from [1], without employing complicated data structures. The details follow below.

## Intersection types

Restated in our terms, Observation 3.1 means that a dark trapezoid  $T' \in D_c^{k-1}$  gets fully or partially lit at step  $k$  if and only if it is intersected by some (different-orientation) trapezoid  $T^* \in S_{c^*}^{k-1}$  lit at step  $k-1$ . We distinguish between two types of trapezoids intersection (Fig. 15).

**Definition 3.3.**  $T', T^*$  are *flush* if a side of  $T'$  overlaps with a side of  $T^*$ . We say that  $T'$  is (fully or partially) *flush-lit* by  $T^*$ .

A flush trapezoid  $T'$  is lit fully or only partially depending on whether its side fully or only partially overlaps with the sides of the trapezoids in  $\bigcup_{c^*} S_{c^*}^{k-1}$  that are flush with  $T'$ .

**Definition 3.4.**  $T', T^*$  *straddle* each other if both bases of  $T'$  intersect both bases of  $T^*$ . We say that  $T'$  is (fully) *straddle-lit* by  $T^*$ .

In particular, if a side of  $T^*$  overlaps with a base of  $T'$  or vice versa, then  $T', T^*$  are counted as straddling, not as flush.

Note that flush intersection and straddling are the only possible ways for two trapezoids from  $D_c^{k-1}, D_{c^*}^{k-1}$  to intersect: because vertices and sides of trapezoids belong to edges of  $P$ , if one trapezoid has a vertex inside the other, then the vertex is on an edge of  $P$  and the trapezoids

are flush. (The flush intersection may be degenerate when the trapezoids just share a common vertex.) Also note that a straddle-lit trapezoid is necessarily fully lit.

With Definitions 3.3 and 3.4, step  $k$  of the algorithm can be completed as follows: Find dark  $c$ -trapezoids flush with trapezoids from  $S_{c^*}^{k-1}$ , and (fully or partially) light them: check for each found flush trapezoid  $T'$  whether it is flush-lit fully or partially. For each partially flush-lit trapezoid determine the dark subtrapezoid  $T'' \subset T'$ , and label with  $k$  the (sub)trapezoid(s) that form  $T' \setminus T''$ . After this has been done for all  $c^* \in C^*$ , i.e., after all flush trapezoids are processed, any dark trapezoid will either fully remain dark in  $D^k$  or will be fully straddle-lit (i.e., there will be no more partial lighting and splitting). So what remains is to straddle-light  $c$ -trapezoids. For that we clip each  $c^*$ -trapezoid  $T^* \in S_{c^*}^{k-1}$  to a  $(c, c^*)$ -parallelogram  $P^*$ , using  $c$ -segments going through vertices of  $T^*$ . We plant  $P^*$  into the  $c$ -trapezoid  $T'$  to which the lower base of the parallelogram belongs and do a sweep analogous to the rectilinear case to discover the trapezoids intersected by the parallelograms.

The clipping-planting-sweeping is repeated for each  $c^* \in C^*$ , i.e., overall, to straddle-light the  $c$ -trapezoids in  $S_c^k$ , we perform  $C - 1$  sweeps, one per  $c^* \in C^*$ . We proceed to a detailed description of the flush- and straddle-lightings.

### Flush-lighting

Sides of trapezoids belong to edges of  $P$ . We say that an edge  $e$  *supports* a trapezoid if one of its sides belongs to  $e$ . We maintain the ordered list  $L_c(e)$  of  $c$ -trapezoids supported by  $e$ . Similarly, we store with each trapezoid the 2 edges of  $P$  that support its sides. The flush-lighting is then done as follows: For every  $c^*$ -trapezoid  $T^* \in S_{c^*}^{k-1}$  and each edge  $e$  that supports  $T^*$ , locate the vertices  $a, b$  of  $T^*$  (lying on  $e$ ) in the list  $L_c(e)$ . All (dark) trapezoids lying between  $a$  and  $b$  are labeled  $k$ . One of the trapezoids  $T'_a, T'_b$  containing  $a, b$  in the interior of the side is marked to be split, at the end of flush-lighting, by a horizontal cut through  $a$  or  $b$ , unless more of the trapezoid is flush-lit by another trapezoid. Refer to Fig. 15, left.

### Straddle-lighting

Now that all flush-lit trapezoids have been found and lit, we have to find straddle-lit trapezoids. Fix  $c^* \in C^*$ .

Clip each  $c^*$ -trapezoid  $T^* \in S_{c^*}^{k-1}$  to the parallelogram  $P^*$  using horizontal lines through vertices of  $T^*$ . Denote the set of the obtained parallelograms by  $S^{\triangleright c^*}$ . Any  $c$ -trapezoid straddled by  $T^*$  is also straddled by  $P^*$ , and thus straddle-lighting with  $c^*$ -trapezoids is equivalent to finding dark trapezoids intersected by parallelograms from  $S^{\triangleright c^*}$ . This can be accomplished with a sweep, called the  $(c, c^*)$ -sweep, similarly to the case  $C = 2$  from Section 3.1. Below we describe the few differences.

Some  $c^*$ -trapezoids from  $S_{c^*}^{k-1}$  (such as, e.g., trapezoid  $T_1^*$  from Fig. 15, right) have lower bases supported by  $c$ -edges of  $P$ . Planting the parallelograms cut out from such trapezoids is identical to the rectilinear case—the pots are read off directly from the trapezoidations  $D_c, D_{c^*}$  augmented with the little auxiliary information as described in Section 3.1. The rest of the trapezoids from  $S_{c^*}^{k-1}$  (such as, e.g., trapezoid  $T_2^*$  from Fig. 15, right, or  $T^*$  from Fig. 15, left) are flush with trapezoids from  $D_c^{k-1}$ . The pot  $T'$  for the parallelogram  $P^*$  cut out from such a trapezoid  $T^*$  can be determined from the list  $L_c(e)$ , where  $e$  is the edge supporting  $T^*$  and  $T'$ : all that is needed is to locate in which trapezoid from the list the vertex of  $T^*$  lands (and if the vertex of  $T^*$  is a common vertex of two  $c$ -trapezoids, then the upper of the two is chosen as the pot). Note that if  $T^*, T'$  are flush, then  $P^*$  may not be planted into  $T'$  “all the way to the bottom”, and so the parallelogram event corresponding to the lower base of  $P^*$  may not coincide

with any trapezoid event; this means that the sweepline does not intersect  $P^*$  at the time when  $T'$  is the event, and hence the pot is not straddle-lit. This is not a problem because the pot is flush-lit by  $T^*$  anyway, and no other  $c$ -trapezoid is intersected by  $P^*$  before the parallelogram comes out of the pot (by which time the intersection of the sweepline with  $P^*$  is already present in the sweepline status). Thus, the only fix to the sweep that we need is unconditional insertion of the (upper) neighbors of  $T'$  into the event queue, even when the lower base of  $T'$  does not intersect the sweepline-status intervals.

The final, minor difference from the rectilinear case is that the parallelograms in  $S^{\succ\circ}$  are not vertical. To account for this, every operation on the sweepline status is preceded by a horizontal shift of the processed interval. Specifically, recall that the operations supported by the interval tree are insertion, deletion, and query of an interval. If the operation is performed when the sweepline is at height  $h$ , we shift the interval by  $h/\tan\alpha$  before the operation, where  $\tan\alpha$  is the slope of  $c^*$ .

We emphasize that clipping by the  $c$ -segments is done only to find  $c$ -trapezoids straddle-lit by  $c^*$ -trapezoids. After the  $(c, c^*)$ -sweep completes, the  $c^*$ -trapezoids are “unclipped” back to what they were (and in general, during a  $(c_1, c_2)$ -sweep,  $c_2$ -trapezoids lit at the previous step are only temporarily clipped into  $(c_1, c_2)$ -parallelograms using  $c_1$ -segments through the vertices).

## Analysis

Any of the  $O(Cn)$  trapezoids in the final  $c$ -maps is either flush-lit or straddle-lit. Flush-lighting takes overall  $O(C^2n \log n)$  time. Indeed, for every trapezoid  $T^*$  that flush-lights  $c$ -trapezoids through an edge  $e$ , it takes  $O(\log n)$  time to locate the vertices  $a, b$  of  $T^*$  (lying on  $e$ ) in the list  $L_c(e)$ . Overall there are  $O(Cn)$  trapezoids  $T^*$ , and for each we have to locate the vertices  $a, b$  in the  $C - 1$  lists  $L_c(e)$ . Thus the locating takes overall  $O(C^2n \log n)$  time. After the vertices  $a, b$  have been located, it takes  $O(n_e)$  time to label each (dark) trapezoid  $T'$  supported by  $e$  (here  $n_e$  is the number of the trapezoids that are flush with  $T^*$ ). Again, overall there are  $O(Cn)$  trapezoids  $T'$ , and each can be flush-lit from at most  $C - 1$  directions. Thus the total time spent in the labeling (not counting the time spent in locating the vertices  $a, b$ ) is  $O(C^2n)$ .

As for straddle-lighting, by the non-degeneracy assumption, any trapezoid has  $O(1)$  neighbors. Thus, processing an event during any of the sweeps involves a constant number of priority queue and/or interval tree operations, i.e.,  $O(\log n)$  time per event. To bound the number of events, observe that exactly as in the rectilinear case, any trapezoid inserted in the event queue at step  $k$  is either itself intersected by a parallelogram from  $S^{\succ\circ}$ , or has a lower neighbor intersected by a parallelogram from  $S^{\succ\circ}$ . Thus, just as in the rectilinear case, any trapezoid enters the event queue on at most 7 consecutive BFS steps. At any step  $k$ , a  $c$ -trapezoid may appear in the event queue during each of the  $C - 1$   $(c, c^*)$ -sweeps. Thus, since there are  $O(Cn)$  trapezoids, we have  $O(C^2n)$  events, and the total running time of straddle-lighting is  $O(C^2n \log n)$ .

As for the space, the interval tree uses  $O(n)$  space at a single sweep; moreover, because we do the sweeps in different directions independently, we never need more than a single interval tree during the execution of the algorithm. The dominating term is thus the storage of the  $C$  trapezoidations, each requiring  $O(n)$ -space.

**Theorem 3.5.** *An  $O(Cn)$ -size data structure can be built to answer  $C$ -oriented link distance queries in  $C$ -oriented domains in  $O(C \log n)$  time; a minimum-link path can be output in additional time proportional to the distance. The preprocessing time and space are  $O(C^2n \log n)$  and  $O(Cn)$ .*

### 3.3 Extensions

We apply our methods to compute minimum-link  $C$ -oriented paths in arbitrarily oriented domains, to give a faster 2-approximation algorithm for finding minimum-link  $C$ -oriented paths, and to quantify what unrestricted-orientation paths are approximable by  $C$ -oriented ones.

#### 3.3.1 $C$ -oriented paths in arbitrary domains

A simple generalization allows us to compute the  $C$ -oriented link distance map also in a domain that is not necessarily  $C$ -oriented. Examining the algorithm from the previous section, we see that orientations of the *sides* of the trapezoids did not play any role in the algorithm (what was important is that the *bases* of the trapezoids are  $C$ -oriented). The only place in which  $P$ 's  $C$ -orientedness was used is the algorithm's analysis: Observation 3.2 and its corollary that the final  $c$ -map is linear-size. That is, even if  $P$  is an arbitrary domain, we can define the  $c$ -map as the decomposition of the free space into trapezoids labeled with the same  $c$ -distance from  $s$ , and by running our algorithm without any modifications we discover the trapezoids in the  $c$ -map one-by-one (i.e., label-by-label), until all of the free space is lit. Denoting by  $N$  the maximum complexity of the  $c$ -map (i.e., the number of trapezoids in the  $c$ -map) over  $c \in C$ , we thus obtain:

**Corollary 3.6.** *An  $O(CN)$ -size data structure can be constructed to answer  $C$ -oriented link distance queries in  $O(C \log N)$  time; a minimum-link path can be output in time proportional to the distance. The preprocessing time and space are  $O(C^2 N \log N)$  and  $O(CN)$ .*

The above result is not entirely satisfactory because if  $P$  is not  $C$ -oriented,  $N$  may be unbounded and even infinite – see, e.g., Fig. 16 (so one would have to assume a model of computation in which operations on large numbers can still be carried out in constant time). This is due to the possibility that a trapezoid from  $D_c^0$ —the initial trapezoidal decomposition—may get split into an unbounded number of subtrapezoids in the final  $c$ -map. When can this happen? If there exists a  $c$ -oriented line, for some  $c \in C$ , that intersects both bases of the trapezoid, then the link distances to different points inside the trapezoid differ only by a constant, and, thus, all of it will be lit during  $O(1)$  consecutive BFS steps—without any modifications of the algorithm. Hence, to avoid the dependence on  $N$ , we need to modify the algorithm only to handle trapezoids whose bases cannot be straddled by a  $C$ -oriented line. We determine such trapezoids in  $D_c^0$  and mark them as *problematic*.

We will declare “deep” portions of problematic trapezoids as separate cells in the link distance map. The path to a query point  $q$  inside such a cell consists of 2 parts: a path from  $s$  to enter the trapezoid, and a “zigzag” of extreme orientations to  $q$ . The number of links in the first part is given by the usual link distance map, and the number of links in the second part can be determined in constant time (assuming constant-time floor function) because of the regular pattern of the path—it bounces off of the sides of the trapezoid until reaching the query point. Overall, we obtain

**Theorem 3.7.** *An  $O(Cn)$ -size data structure can be constructed to answer  $C$ -oriented link distance queries in arbitrary domains in  $O(C \log n)$  time; a minimum-link path can be output in time proportional to the distance. The preprocessing time and space are  $O(C^2 n \log n)$  and  $O(Cn)$ .*

The proof of the theorem can be found in Appendix B.

#### 3.3.2 Approximate $C$ -oriented paths

The  $C$ -oriented link distance can be 2-approximated by requiring that every second link of the path is horizontal. To find a minimum-link  $C$ -oriented path with this requirement one can do

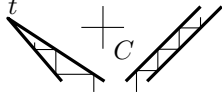


Figure 16:  $C$ -oriented min-link paths fail to approximate min-link paths with unrestricted orientations. Left:  $t$  is not reached with finitely many links. Right: The number of  $C$ -oriented links required to pass through a corridor may depend on its width rather than on its complexity.

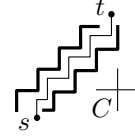


Figure 17: Even when the domain is  $C$ -oriented, a path having  $O(1)$  links of arbitrary orientation can require  $\Omega(n)$   $C$ -oriented links.

the BFS in the trapezoids intersection graph with one modification: instead of checking for intersection between trapezoids for all pairs of orientations, only check for intersections between the horizontal trapezoids and the other ones. Such a modification decreases the running time of our algorithm to  $O(Cn \log n)$ ; the space remains  $O(Cn)$ , since the  $C$  trapezoidations of  $P$  are still constructed.

To reduce the space to  $O(n)$  we do only the horizontal trapezoidation, and go back to the rectilinear-case ideas of Das and Narasimhan [5] who do the labeling of horizontal trapezoids *without* using the vertical ones (in our case, we label the horizontal trapezoids without using *any* other ones). In particular, without the other trapezoidations we cannot use our planting (there is simply nothing to plant!) to initialize the sweep. Thus we do both the UpSweep and the DownSweep as in [5], starting from trapezoids labeled on the previous step.

Overall, we obtain

**Theorem 3.8.** *An  $O(n)$ -size data structure can be constructed to answer  $C$ -oriented link distance queries to within a multiplicative error of 2; that is, if the minimum number of links in a  $C$ -oriented path from  $s$  to the query point  $q$  is  $k$ , the data structure will report a number  $l \leq 2k$ . The query time is  $O(\log n)$ , and a  $C$ -oriented  $l$ -link  $s$ - $q$  path can be output in additional  $O(l)$  time. The preprocessing time and space are  $O(Cn \log n)$  and  $O(n)$ .*

The proof of the theorem can be found in Appendix C.

### 3.3.3 Approximating paths with “robust” edges

We would like to use  $C$ -oriented paths to approximate the link distance of paths with unrestricted orientations. Unfortunately, as Fig. 16 (cf. [1, Fig. 1]) illustrates, the number of links in a minimum-link  $C$ -oriented path may be much higher than that of a path with unrestricted orientations: there are geometric configurations in which an unrestricted path of one link must be replaced by many, even infinitely many,  $C$ -oriented links. Even if we restrict the obstacle boundaries to use the same  $C$  orientations (i.e., we work in a  $C$ -oriented domain), Fig. 17 illustrates that an unrestricted path of one link may require  $\Omega(n)$   $C$ -oriented links. To use  $C$ -oriented paths to approximate link distances, we define a model of paths that we call “robust” paths.<sup>3</sup>

One feature is common to examples that show poor approximation power of  $C$ -oriented paths (e.g., Figs. 16,17): when an unrestricted path can have substantially fewer links than a  $C$ -oriented path, the set of orientations of a critical link of the unrestricted path is very narrow.

<sup>3</sup>Note the contrast with *length* approximation: in terms of (Euclidean) length,  $C$ -oriented paths approximate general paths with additive relative error of  $O(1/C^2)$ , irrespective of whether the domain is  $C$ -oriented or not.

This is also the case in instances of the minimum-link problem used in showing 3SUM-hardness (Fig. 2)—an optimal path must “shoot” very precisely through free space. That is, a small deviation in the orientation of an edge in a minimum-link path renders the edge—and hence the whole path—infeasible.

We envision that such non-robustness of minimum-link paths with unrestricted orientations is highly undesirable in applications: if a path is to be followed by a robot, one would not want the robot to hit an obstacle due to a small deviation in the steering direction; if the path edges are communication links, the communication should not be interrupted due to a small error in the direction; etc. We thus quantify edge robustness as follows: for  $\varphi \geq 0$ , an edge  $pq$  is  $\varphi$ -robust if the isosceles triangle with altitude  $pq$  and angle  $2\varphi$  at the apex  $p$  does not intersect obstacles (Fig. 18). A path is  $\varphi$ -robust if all of its edges are  $\varphi$ -robust. (We use triangles centered on edges instead of circular sectors for technical reasons.) The definition is directional –  $pq$  can be  $\varphi$ -robust while  $qp$  is not; one can modify the definition to make it symmetric. Note that according to our definition, robustness increases with  $\varphi$ ; 0-robust is “not at all robust”. Note also that adding vertices in the middle of a path’s edges gives a new path whose robustness is at least that of the original (Fig. 18, right). Intuitively this is fair: say, instead of sending a robot in a single command through a narrow corridor, with a single link, one may guide the robot through the corridor more carefully by adding extra reference points (path vertices) along the way.

We postulate that for  $C$ -oriented paths there is no issue of “wiggling” of edges directions. That is, unlike with the unrestricted-orientations paths, if a  $C$ -oriented path is computed, it can actually be used with no fear of it becoming infeasible due to an error in edge orientation. The justification of such an assumption is twofold. First, from the theoretical point of view  $C$ -oriented paths are “discrete” and allow for no continuous change of edge orientation. Second, we are inspired by real-world mechanisms design: to set the orientation for an unrestricted path, one may need to turn a knob/throttle—this may not be easy to do with perfect precision. On the other hand, for  $C$ -oriented paths, the robot’s wheels turning angle or the communication direction can be set with essentially no error. This can be achieved with any mechanism in which the shaft is connected to a gear that may rest against a notch—the gear is in a static equilibrium only when turned by a multiple of its angle of action. An example of such a mechanism is a ratchet (Fig. 19); many others exist [15]—facing crown gears, socket on a bolt, external gear fitting inside an external gear, etc.

Even though in general  $C$ -oriented paths do not approximate paths with unrestricted orientations at all (as seen in Figs. 16,17), as the next lemma shows,  $C$ -oriented paths approximate *robust* paths well. Specifically, let  $\varphi = 180/i$ , for some integer  $i$ , and let  $C_\varphi$  be the set of orientations evenly spaced along the unit circle with angle  $\varphi$  between consecutive orientations.

**Lemma 3.9.** *If there exists a  $k$ -link  $\varphi$ -robust  $s$ - $t$  path  $\pi$ , then there exists a  $(k + 1)$ -link  $C_\varphi$ -oriented  $s$ - $t$  path  $\pi'$ .*

*Proof.* For an edge  $p_i p_{i+1}$  of a  $k$ -link path  $\pi = (s, p_1, p_2, \dots, p_{k-1}, p_k = t)$  let  $T_{p_i}$  be the isosceles triangle with height  $p_i p_{i+1}$  and angle  $2\varphi$  at  $p_i$ . Refer to Fig. 20. Since  $\pi$  is  $\varphi$ -robust, the triangle is obstacle-free and contains a  $C_\varphi$ -oriented segment connecting  $p_i$  to the base of the triangle on each side of  $p_i p_{i+1}$ . Denote these *snapped* segments by  $p_i p_i^+$ ,  $p_i p_i^-$ . Let  $q_k$  be the point within  $T_{p_{k-1}}$  where the line through  $t$  that is parallel to  $p_{k-1} p_{k-1}^-$  intersects  $p_{k-1} p_{k-1}^+$ . Now, the line through  $p_{k-1}$  and  $q_k$  must cross one of the two snapped segments through  $p_{k-2}$  ( $p_{k-2} p_{k-2}^+$  or  $p_{k-2} p_{k-2}^-$ ). Let  $q_{k-1}$  be the point of crossing. (In fact, the line through  $p_{k-1}$  and  $q_k$  must intersect *exactly one* of the two snapped segments, unless it is perpendicular to  $p_{k-2} p_{k-1}$  (meaning it contains the base,  $p_{k-2} p_{k-2}^+$ , of  $T_{k-2}$ ), in which case we select  $q_{k-1}$  to be either point  $p_{k-2}^-$  or  $p_{k-2}^+$ .) Note that the segment  $q_{k-1} q_k$  lies inside  $T_{k-2} \cup T_{k-1}$  and is, therefore, within  $P$ .



the 3SUM-hardness construction, and we thank Haitao Wang for discussions on several aspects of the paper, including the remark that one can test in quadratic time if the  $s$ - $t$  link distance is at most 3. J. Mitchell is partially supported by the National Science Foundation (CCF-1018388), Metron Aviation, and NASA Ames. V. Polishchuk is funded by the Academy of Finland grant 1138520. M. Sysikaski is funded by the Research Funds of the University of Helsinki.

## References

- [1] J. Adegeest, M. H. Overmars, and J. Snoeyink. Minimum-link  $c$ -oriented paths: Single-source queries. *International Journal of Computational Geometry and Applications*, 4(1):39–51, 1994.
- [2] B. Aronov, S. Har-Peled, and M. Sharir. On approximate halfspace range counting and relative epsilon-approximations. In *Proceedings of the 23rd Annual Symposium on Computational Geometry*, pages 327–336. ACM, 2007.
- [3] B. Chazelle, H. Edelsbrunner, M. Grigni, L. J. Guibas, J. Hershberger, M. Sharir, and J. Snoeyink. Ray shooting in polygons using geodesic triangulations. *Algorithmica*, 12(1):54–68, 1994.
- [4] B. Chazelle and E. Welzl. Quasi-optimal range searching in space of finite vc-dimension. *Discrete & Computational Geometry*, 4:467–489, 1989.
- [5] G. Das and G. Narasimhan. Geometric searching and link distance. In *WADS’91*, pages 261–272, 1991.
- [6] M. de Berg. On rectilinear link distance. *Computational Geometry: Theory and Applications*, 1:13–34, 1991.
- [7] E. D. Demaine, J. S. B. Mitchell, and J. O’Rourke. The open problems project. <http://maven.smith.edu/~orourke/TOPP/>.
- [8] A. Gajentaan and M. H. Overmars. On a class of  $O(n^2)$  problems in computational geometry. *Computational Geometry: Theory and Applications*, 5:165–185, 1995.
- [9] S. K. Ghosh. Computing the visibility polygon from a convex set and related problems. *Journal of Algorithms*, 12(1):75–95, 1991.
- [10] J. E. Goodman and J. O’Rourke. *Handbook of Discrete and Computational Geometry*. CRC Press series on discrete mathematics and its applications. Chapman & Hall/CRC, 2004.
- [11] R. Güting. *Conquering Contours: Efficient Algorithms for Computational Geometry*. PhD thesis, Fachbereich Informatik, Universität Dortmund, 1983.
- [12] R. H. Güting and T. Ottmann. New algorithms for special cases of the hidden line elimination problem. *Comp. Vis., Graph., Image Proc.*, 40(2):188–204, 1987.
- [13] J. Hershberger and J. Snoeyink. Computing minimum length paths of a given homotopy class. *Computational Geometry: Theory and Applications*, 4:63–97, 1994.
- [14] J. Hershberger and S. Suri. A pedestrian approach to ray shooting: Shoot a ray, take a walk. *Journal of Algorithms*, 18(3):403–431, 1995.



- [15] G. D. Hiscox. *Mechanical Movements, Powers And Devices*. Kessinger Publishing, LLC, 2007.
- [16] H. Imai and T. Asano. Dynamic segment intersection search with applications. In *FOCS*, pages 393–402, 1984.
- [17] H. Imai and T. Asano. Efficient algorithms for geometric graph search problems. *SIAM J. Comput.*, 15(2):478–494, 1986.
- [18] H. Imai and T. Asano. Dynamic orthogonal segment intersection search. *Journal of Algorithms*, 8(1):1–18, 1987.
- [19] S. Kahan and J. Snoeyink. On the bit complexity of minimum link paths: Superquadratic algorithms for problems solvable in linear time. *Computational Geometry: Theory and Applications*, 12(1-2):33–44, 1999.
- [20] H. Kaplan, J. Matousek, and M. Sharir. Simple proofs of classical theorems in discrete geometry via the guth-katz polynomial partitioning technique. *Discrete & Computational Geometry*, 48(3):499–517, 2012.
- [21] D. T. Lee. Personal communication.
- [22] D. T. Lee, C. D. Yang, and C. K. Wong. Rectilinear paths among rectilinear obstacles. *Discrete Appl. Math.*, 70:185–215, 1996.
- [23] A. Maheshwari, J.-R. Sack, and D. Djidjev. Link distance problems. In J.-R. Sack and J. Urrutia, editors, *Handbook of Comp. Geom.*, pages 519–558. Elsevier, 2000.
- [24] J. Matoušek. *Geometric Discrepancy: An Illustrated Guide*. Algorithms and Combinatorics. Springer, 1999.
- [25] J. Mitchell, G. Rote, and G. Woeginger. Minimum-link paths among obstacles in the plane. *Algorithmica*, 8(1):431–459, 1992.
- [26] J. S. B. Mitchell. Geometric shortest paths and network optimization. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 633–701. Elsevier Science B.V. North-Holland, Amsterdam, 2000.
- [27] G. Neyer. Line simplification with restricted orientations. In F. K. H. A. Dehne, A. Gupta, J.-R. Sack, and R. Tamassia, editors, *WADS’99*, pages 13–24, 1999.
- [28] T. Ohtsuki. Gridless routers - new wire routing algorithm based on computational geometry. In *International Conference on Circuits and Systems, China*, 1985.
- [29] T. Ohtsuki and M. Sato. Gridless routers for two layer interconnection. In *IEEE Int. Conf. Comput. Aided Design*, pages 76–78, 1984.
- [30] V. Polishchuk and M. Sysikaski. Faster algorithms for minimum-link paths with restricted orientations. In *WADS’11*, pages 655–666, 2011.
- [31] G. J. E. Rawlins and D. Wood. Optimal computation of finitely oriented convex hulls. *Information and Computation*, 72(2):150–166, 1987.
- [32] M. Sato, J. Sakanaka, and T. Ohtsuki. A fast line-search method based on a tile plane. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 588–591. IEEE, 1987.

- [33] S. Suri. A linear time algorithm with minimum link paths inside a simple polygon. *Computer Vision, Graphics and Image Processing*, 35(1):99–110, 1986.
- [34] S. Suri and J. O'Rourke. Worst-case optimal algorithms for constructing visibility polygons with holes. In *Proceedings of the 2nd Annual Symposium on Computational Geometry*, pages 14–23, New York, NY, USA, 1986. ACM.
- [35] E. Welzl. On spanning trees with low crossing numbers. In B. Monien and T. Ottmann, editors, *Data Structures and Efficient Algorithms*, volume 594 of *Lecture Notes in Computer Science*, pages 233–249. Springer, 1992.
- [36] P. Widmayer, Y.-F. Wu, and C. K. Wong. On some distance problems in fixed orientations. *SIAM Journal on Computing*, 16(4):728–746, 1987.
- [37] C. D. Yang, D. T. Lee, and C. K. Wong. On bends and lengths of rectilinear paths: a graph theoretic approach. *International Journal of Computational Geometry and Applications*, 2(1):61–74, 1992.
- [38] C. D. Yang, D. T. Lee, and C. K. Wong. On bends and distances of paths among obstacles in 2-layer interconnection model. *IEEE Transactions on Computing*, 43(6):711–724, 1994.
- [39] C. D. Yang, D. T. Lee, and C. K. Wong. Rectilinear paths problems among rectilinear obstacles revisited. *SIAM Journal on Computing*, 24:457–472, 1995.

## Appendix

### A A note on a claim in [38]

The abstract of [38] announces an optimal  $O(n \log n)$ -time  $O(n)$ -space algorithm to find a minimum-link path (a minimum-*bend* path, or MBP, in terminology of [38]): “optimal  $\theta(e \log e)$  time algorithms are presented to find the shortest path and the minimum-bend path using linear space”. We did not find the MBP considered anywhere in the paper, though. Moreover, the penultimate paragraph of the Introduction (p. 712) states that “problems MBP and yD-SP have been solved optimally in [5], [11], [24], [25]”. The references [5], [11], [24], [25] from [38] are our [5, 16, 28, 29]. However, Imai and Asano [16–18] claimed  $O(n \log n)$  time *and* space for their method. Also, Ohtsuki’s algorithm [28] runs in  $O(n \log^2 n)$  time and  $O(n)$  space (as acknowledged in the second paragraph of the Introduction [38, p. 711]). It is possible that, at the time of writing [38], only the time bound was considered, and not the space bound, in describing the “optimality” of the algorithm. Unfortunately, too much time passed to resolve the possible confusion now [21].

### B Handling problematic trapezoids

Recall that a trapezoid is *problematic* if its bases cannot be straddled by a  $C$ -oriented line.

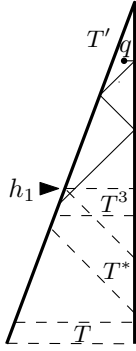


Figure 21: A zigzag through  $T'$ . The labels of  $T, T^*, T^3$  are  $k, k+2, k+3$ , respectively.

Let  $T'$  be a problematic  $c$ -trapezoid. As in Section 3.2, assume the orientation  $c \in C$  is horizontal and implicit, and  $C^* = C \setminus c$ . We first describe how to handle  $T'$  when it is a triangle, i.e., when one of its bases has length 0 (Fig. 21). Without loss of generality, the 0-length base is the upper base of  $T'$ . Let  $c_1, c_2 \in C$  be the orientations closest to those of the sides of  $T'$ ; we refer to orientations  $c_1$  and  $c_2$  as *extreme* orientations. We say that a path  $\pi'$  from a point  $q \in T'$  is a *zigzag* if it starts with a horizontal segment and its vertices bounce between the sides of  $T'$  using extreme-orientation links (two zigzags start at  $q$ , one going left, the other going right). Let  $S_1, S_2$  denote the sides of  $T'$ . Given two points  $h_1 \in S_1, h_2 \in S_2$ , the following query can be answered in constant time (assuming constant-time floor function): What is the minimum number of links in a zigzag from  $q$  needed to reach  $S_1$  below  $h_1$  or to reach  $S_2$  below  $h_2$ ? Denote the answer to the query by  $K(q, h_1, h_2)$ .

Let  $k$  be the BFS step at which  $T'$  is first partially lit. Continue the BFS for 3 more steps. Let  $T^3 \subset T'$  be the label- $(k+3)$   $c$ -trapezoid, and let  $h_1 \in S_1, h_2 \in S_2$  be the highest points on the sides of  $T'$  reached by the extreme-orientations label- $(k+2)$  trapezoids. We declare the part of  $T'$  above  $T^3$  to be a separate single cell in the  $c$ -map. The  $c$ -distance to a query point  $q$  in the cell is given by  $k+2+K(q, h_1, h_2)$ .

To justify the correctness of our approach, recall from Section 3.2 that only flush-lighting leads to trapezoid splitting (straddle-lighting does not). This means that the subtrapezoid  $T \subset T'$  lit at step  $k$  was flush-lit. Similarly, every subtrapezoid  $T'' \subset T'$  whose label  $k''$  is larger than  $k$  is flush-lit. Consider a subtrapezoid  $T'' \subset T'$  with label  $k'' \geq k+3$ . Let  $T^*$  be the  $c^*$ -trapezoid with label  $k''-1$  that (flush-)lit  $T''$ . It appears that  $T^*$  was flush-lit itself, for otherwise its side must have been supported outside  $T'$ , but then  $T^*$  must have intersected with the label- $k$  trapezoid  $T$ —a contradiction to  $k''-1 \geq k+2$ .

The crucial observation about flush-lighting is that a minimum-link path to a point  $q$  inside

a flush-lit trapezoid without loss of generality bounces off of the trapezoid side (i.e., the path's last turn before  $q$  without loss of generality belongs to a side of the trapezoid). Hence, a minimum-link  $c$ -path  $\pi'$  from a point  $q \in T''$  without loss of generality bounces off of the sides of  $T'$  (at least) until reaching a label- $(k+2)$  trapezoid. By a local modification argument, it does not hurt to make all non-horizontal, bouncing links of  $\pi'$  extreme. Thus, without loss of generality, the part of  $\pi'$  reaching an extreme label- $(k+2)$  trapezoid is a zigzag, which proves the formula for the link distance to  $q$ .

We now describe the modifications of the procedure to handle a problematic trapezoid  $T'$  whose bases are each of positive length. Exactly as above, we find  $h_1 \in S_1$  and  $h_2 \in S_2$ , the highest points on the sides of  $T'$  reached by the extreme-orientations label- $(k+2)$  trapezoids. We then “propagate” zigzags from  $h_1$  and  $h_2$  up to the other end of  $T'$ . Specifically, let  $T^*$  be the  $c^*$ -trapezoid, supported by a side of  $T'$ , that penetrates deepest into  $T'$  from its upper base ( $T^*$  can be read from the support list of the edge supporting the side). In constant time we can determine the minimum number  $K^*$  of links necessary to reach  $T^*$  from  $h_1$  or  $h_2$ . We assign *temporary* label  $k+2+K^*$  to  $T^*$ . If  $T^*$  is not lit by BFS step  $k+2+K^*$ , the label becomes permanent, and  $T^*$  is inserted into  $S_{c_i}^{k+2+K^*}$ . Otherwise, if  $T^*$  is lit before the step  $k+2+K^*$ , we propagate the zigzag path from it down through  $T'$ . In constant time we determine where the path meets with zigzags from  $h_1, h_2$  and establish a “bisector” trapezoid  $T_b \subset T'$  in the  $c$ -map; the bisector can be reached through either base of  $T'$ . The portions of  $T'$  below and above the bisector become separate cells in the map, just as in the case of a degenerate, triangular trapezoid described above.

In comparison with Section 3.2, we do  $O(C)$  additional work per trapezoid in  $D^0$ . Thus our time and space bounds of Theorem 3.5 carry over to the case of  $C$ -oriented paths in arbitrarily oriented domains.

## C Approximating $C$ -oriented paths

We are 2-approximating the  $C$ -oriented link distance by requiring that every second link of the path is horizontal. To find a minimum-link  $C$ -oriented path with this requirement, it is enough to do only the horizontal trapezoidation, and do a BFS in it (again, starting from the maximal free-space horizontal segment though  $s$ ): label with  $k+2$  the unlabeled trapezoids that are seen, in at least one direction  $c^* \in C$ , from a label- $k$  trapezoid. This is *exactly* the problem solved by Das and Narasimhan [5] (for the case when  $c^*$  is vertical). Thus, for each orientation  $c^*$  we can march through the decomposition with a  $c^*$ -UpSweep and a  $c^*$ -DownSweep of [5]: just as [5], we do not need the  $c^*$ -trapezoidation for that. The only difference is that when  $c^*$  is not vertical, some trapezoids may be lit only partially, but this is easy to keep track of by maintaining, for each side of every trapezoid, the maximum and minimum height reached by the  $c^*$ -rays coming from label- $k$  trapezoids.

To give more details, we first recap the sweep algorithm of Das and Narasimhan [5]. Recall that the goal of the UpSweep is as follows: Given a set  $S$  of trapezoids (labeled  $k$ ), label with  $k+2$  the trapezoids that are seen by looking up from  $S$ . The sweepline starts at  $-\infty$ , with empty status, and moves upward. The sweepline status is the set of points on the line that are seen by looking up from  $S$ . Thus, the status is a set of disjoint intervals, which can be kept in a simple structure like the one in Section 3.1 (we see no need to use a complicated structure to maintain some “fronts” of merging and splitting “windows” as proposed in [5]).

The events are bases of trapezoids. The events are ordered by height. In case of ties, upper bases are processed before lower bases. Initially, the queue contains the trapezoids  $S$ .

Processing of the lower base of a trapezoid  $T \in S$  is simple: the base is added to the sweepline status. Processing of the upper base of a trapezoid  $T \in S$  involves the following: (1) the part

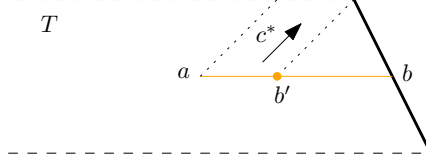


Figure 22: As the sweepline moves up, the sweepline status interval  $ab$  changes continuously, as  $b$  slides along the side of  $T$ . However, inside  $T$  the interval does not intersect any other trapezoid. Thus, as soon as the sweepline enters  $T$ ,  $ab$  can be clipped to  $ab'$ ; there will be no undetected intersections (no false-negatives).

of the base that rests against an obstacle (if any) is removed from the status; (2) bases of the upper neighbors of  $T$  enter the queue.

Processing of the lower base of a trapezoid  $T' \notin S$  is as follows: if the base does not overlap with the sweepline status, the upper base of  $T'$  is removed from the queue; otherwise,  $T'$  is labeled  $k + 2$ . Processing of the upper base of a trapezoid  $T' \notin S$  happens, therefore, only if  $T'$  is labeled  $k + 2$  and involves the same steps as processing the upper base of a trapezoid  $T \in S$ : (1) the part of the base that rests against an obstacle (if any) is removed from the status; and, (2) bases of the upper neighbors of  $T$  enter the queue.

The DownSweep is analogous.

*Remark 6.* Not surprisingly, there are clear parallels between (the recap of) Das and Narasimhan's algorithm given here and our planting-based, upsweep-only algorithm from Section 3.1. Both algorithms use the same idea of labeling the trapezoids step-by-step. Moreover, technically, the sweepline status and the event queue are the same in both algorithms. The only formal difference is how the event queue is initialized: with the aid of planting (as in Section 3.1) or just by  $S$  (here). The conceptual difference is whether the vertical trapezoidation is used explicitly (as in Section 3.1) or not. We extended both ideas to the general case of  $C \geq 2$ . The former appeared more useful for computing exact minimum-link  $C$ -oriented paths (Section 3.2); the latter is useful in achieving our goal here of 2-approximating a  $C$ -oriented path in  $O(n)$  space, without building the other  $C - 1$  trapezoidations.

In the general case (when the domain is not rectilinear and  $c^*$  is not vertical) the goal of the UpSweep is to label with  $k + 2$  the trapezoids that are seen by looking from  $S$  in direction  $c^*$ . Analogously to the rectilinear case, the sweepline status is the set of points on the sweepline that are seen by looking up in orientation  $c^*$ . The issue now is that if a status interval  $I$  touches a side of a trapezoid  $T$ , then the status changes continuously (Fig. 22). Nevertheless, since  $T$  is obstacle-free,  $I$  cannot intersect any other trapezoid inside  $T$ . Thus, as soon as the sweepline enters  $T$ , we clip  $I$  to what it should be after the sweepline exits  $T$ .

Another change we need for the lower-base events is due to the fact that a trapezoid may get only partially lit; thus, we need to find the highest point of  $T$  touched by the intervals. This is easy to do by looking at the intervals that we clip away. After finishing all of the sweeps we split partially lit trapezoids as in Section 3.2.

The described clipping of the sweepline status on a lower-base event and the trapezoid splitting are the only differences from the rectilinear case. Upper-base events are handled exactly as before.